Operating Systems - Assignment

COMP2006

Jordan Yeo
17727626

Curtin University
Science and Engineering

Contents

Introduction

The following report is for the Operating Systems Assignment for 2017. It will detail how mutual exclusion was achieved for processes and threads. Also the processes and threads that accessed shared resources.

1 Process MSSV

For this implementation of the solution, mutual exclusion is achieved by having the parent (consumer) wait for all child (producer) processes to complete execution before continuing. This waiting is implemented by having the parent wait for the semParent semaphore before accessing the shared variable, representing the number of child processes finished. Once the parent has finished creating children it will only be reading the value of the shared variable. The semParent semaphore is to prevent the critical section problem. So the child does not write to the variable as the parent reads. Once a child completes execution it waits for the semParent semaphore before acquiring a mutex lock to update a shared resource, indicating it is finished. The parent is forced to wait until the value of the shared variable shows that all children are finished.

$$wait(semParent)$$
$$// \ critical \ section$$
$$signal(semParent)$$

To ensure mutual exclusion for the shared resources (buffer1, buffer2 and counter) the child waited to acquire a mutex lock before entering its critical section to modify the shared resources. Since buffer1 was never modified and only read from this did not require a mutex lock to be obtained before reading.

$$wait(semMutex)$$
$$// \ critical \ section$$
$$signal(semMutex)$$

The semaphores required (semMutex and semParent), buffer1, buffer2 and counter were implemented using shared memory. The following POSIX shared memory functions were used to create shared memory, and the corresponding functions to close the shared memory:

$$shm\_link()$$
$$ftruncate()$$
$$mmap()$$
$$shm\_unlink()$$
$$close()$$
$$munmap()$$

Zombie processes were killed with the use of signal(SIGCHILD, SIG_IGN).

2 Thread MSSV

To achieve mutual exclusion in the multi-threaded program, the parent (consumer) must wait for all threads (producer) to complete execution. The parent uses *pthread_lock_mutex()* to lock the mutex then performs a *pthread_cond_wait()* on a condition variable. The condition variable represents how many threads are still executing.

Once a thread completes execution, it acquires a mutex lock to alter the condition variable signaling it has finished its task. The thread also performs a conditional wait in the case where the parent is using the shared variable.

$$pthread\_mutex\_lock(\&mutex)$$

$$pthread\_cond\_wait(\&use, \&mutex)$$

$$pthread\_cond\_signal(\&use);$$

$$pthread\_mutex\_unlock(\&mutex)$$

In thread MSSV the shared resources are declared as global variables. Threads have access to the variables declared global in the parent. Before a thread could access the shared resources it would first need to acquire a mutex lock. The function *pthread_lock_mutex()* blocks the caller if the mutex is in use by another. It can then alter the shared resources, counter and buffer2, before releasing the mutex lock.

To allocate the memory for buffer1, buffer2, counter and regions *malloc()* was used. The appropriate *free()* calls were used to free the allocated memory. This was done to ensure memory leaks were not present in the operation of the program.

3 Testing

3.1 Method

To test each implementation of MSSV worked as intended multiple input files were used. With these input files, multiple delays were chosen as well. Input files of various 9x9 numbers were used. Input files were used that were valid, contained one error, contained multiple errors. Also tested were smaller grids than 9x9 and empty files.

Testing was performed on the lab machines in various rooms of Building 314, Level 2.

3.2 Errors

There are no known errors in the process MSSV and the thread MSSV. Care has been taken to ensure potential memory leaks are prevented, by using the appropriate measures to free allocated memory. Memory leaks are not present in the testing of each MSSV currently performed.

3.3 Input Files

$$specTest = \begin{matrix} 6 & 2 & 4 & 5 & 3 & 9 & 1 & 8 & 7 \\ 5 & 1 & 9 & 7 & 2 & 8 & 6 & 3 & 4 \\ 8 & 3 & 7 & 6 & 1 & 4 & 2 & 9 & 5 \\ 1 & 4 & 3 & 8 & 6 & 5 & 7 & 2 & 9 \\ 9 & 5 & 8 & 2 & 4 & 7 & 3 & 6 & 1 \\ 7 & 6 & 2 & 3 & 9 & 1 & 4 & 5 & 8 \\ 3 & 7 & 1 & 9 & 5 & 6 & 8 & 4 & 2 \\ 4 & 9 & 6 & 1 & 8 & 2 & 5 & 7 & 3 \\ 2 & 8 & 5 & 4 & 7 & 3 & 9 & 1 & 6 \end{matrix}$$

$$testFail = \begin{matrix} \boxed{2} & 2 & 4 & 5 & 3 & 9 & 1 & 8 & 7 \\ 5 & 1 & 9 & 7 & 2 & 8 & 6 & 3 & 4 \\ 8 & 3 & 7 & 6 & 1 & 4 & 2 & 9 & 5 \\ 1 & 4 & 3 & 8 & 6 & 5 & 7 & 2 & 9 \\ 9 & 5 & 8 & 2 & 4 & 7 & 3 & 6 & 1 \\ 7 & 6 & 2 & 3 & 9 & 1 & 4 & 5 & 8 \\ 3 & 7 & 1 & 9 & 5 & 6 & 8 & 4 & 2 \\ 4 & 9 & 6 & 1 & 8 & 2 & 5 & 7 & 3 \\ 2 & 8 & 5 & 4 & 7 & 3 & 9 & 1 & 6 \end{matrix}$$

$$multiFail = \begin{matrix} \boxed{2} & 2 & 4 & 5 & 3 & 9 & 1 & 8 & 7 \\ 5 & 1 & 9 & 7 & 2 & 8 & 6 & 3 & 4 \\ 8 & 3 & 7 & 6 & 1 & 4 & 2 & 9 & 5 \\ 1 & 4 & 3 & 8 & 6 & 5 & 7 & 2 & 9 \\ 9 & 5 & 8 & 2 & 4 & 7 & 3 & 6 & \boxed{3} \\ 7 & 6 & 2 & 3 & 9 & 1 & 4 & 5 & 8 \\ 3 & 7 & 1 & 9 & 5 & 6 & 8 & 4 & 2 \\ 4 & 9 & 6 & 1 & 8 & 2 & 5 & 7 & 3 \\ 2 & 8 & 5 & 4 & 7 & 3 & 9 & 1 & 6 \end{matrix}$$

3.4 Expected Results

specTest

```
row 1 is valid
row 2 is valid
row 3 is valid
row 4 is valid
row 5 is valid
row 6 is valid
row 7 is valid
row 8 is valid
row 9 is valid
9 out of 9 columns are valid
9 out of 9 sub-grids are valid

There are 27 valid sub-grids, and thus the solution is valid
```

testFail

```
row 1 is invalid
row 2 is valid
row 3 is valid
row 4 is valid
row 5 is valid
row 6 is valid
row 7 is valid
row 8 is valid
row 9 is valid
8 out of 9 columns are valid
8 out of 9 sub-grids are valid

There are 24 valid sub-grids, and thus the solution is invalid
```

multiFail

```
row 1 is invalid
row 2 is valid
row 3 is valid
row 4 is valid
row 5 is invalid
row 6 is valid
row 7 is valid
row 8 is valid
row 9 is valid
7 out of 9 columns are valid
7 out of 9 sub-grids are valid

There are 21 valid sub-grids, and thus the solution is invalid
```

3.5 Actual Results

Figure 1: Processes: specTest

```
[17727626@lab221-c01 partA]$ ./mssv ../testFiles/specTest.txt 10
Validation result from process ID-19029: row 1 is valid
Validation result from process ID-19030: row 2 is valid
Validation result from process ID-19031: row 3 is valid
Validation result from process ID-19032: row 4 is valid
Validation result from process ID-19033: row 5 is valid
Validation result from process ID-19034: row 6 is valid
Validation result from process ID-19035: row 7 is valid
Validation result from process ID-19036: row 8 is valid
Validation result from process ID-19037: row 9 is valid
Validation result from process ID-19038: 9 out of 9 columns are valid
Validation result from process ID-19039: 9 out of 9 sub-grids are valid
There are 27 valid sub-grids, and thus the solution is valid
```

Figure 2: Threads: specTest

```
[17727626@lab221-b01 partB]$ ./mssv ../testFiles/specTest.txt 10
Validation result from thread ID-2979485440: row 1 is valid
Validation result from thread ID-2971092736: row 2 is valid
Validation result from thread ID-2962700032: row 3 is valid
Validation result from thread ID-2954307328: row 4 is valid
Validation result from thread ID-2945914624: row 5 is valid
Validation result from thread ID-2937521920: row 6 is valid
Validation result from thread ID-2929129216: row 7 is valid
Validation result from thread ID-2920736512: row 8 is valid
Validation result from thread ID-2912343808: row 9 is valid
Validation result from thread ID-2903951104: 9 out of 9 columns are valid
Validation result from thread ID-2895558400: 9 out of 9 sub-grids are valid
There are 27 valid sub-grids, and thus the solution is valid
```

Figure 3: Processes: testFail

```
[17727626@lab221-c01 partA]$ ./mssv ../testFiles/testFail.txt 10
Validation result from process ID-19053: row 1 is invalid
Validation result from process ID-19054: row 2 is valid
Validation result from process ID-19055: row 3 is valid
Validation result from process ID-19056: row 4 is valid
Validation result from process ID-19057: row 5 is valid
Validation result from process ID-19058: row 6 is valid
Validation result from process ID-19059: row 7 is valid
Validation result from process ID-19060: row 8 is valid
Validation result from process ID-19061: row 9 is valid
Validation result from process ID-19062: 8 out of 9 columns are valid
Validation result from process ID-19063: 8 out of 9 sub-grids are valid
There are 24 valid sub-grids, and thus the solution is invalid
```

Figure 4: Threads: testFail

```
[17727626@lab221-b01 partB]$ ./mssv ../testFiles/testFail.txt 10
Validation result from thread ID-2959283968: row 1 is invalid
Validation result from thread ID-2950891264: row 2 is valid
Validation result from thread ID-2942498560: row 3 is valid
Validation result from thread ID-2934105856: row 4 is valid
Validation result from thread ID-2925713152: row 5 is valid
Validation result from thread ID-2917320448: row 6 is valid
Validation result from thread ID-2908927744: row 7 is valid
Validation result from thread ID-2900535040: row 8 is valid
Validation result from thread ID-2892142336: row 9 is valid
Validation result from thread ID-2883749632: 8 out of 9 columns are valid
Validation result from thread ID-2875356928: 8 out of 9 sub-grids are valid
There are 24 valid sub-grids, and thus the solution is invalid
```

Figure 5: Processes: multiFail

```
[17727626@lab221-c01 partA]$ ./mssv ../testFiles/multiFail.txt 10
Validation result from process ID-19070: row 1 is invalid
Validation result from process ID-19071: row 2 is valid
Validation result from process ID-19072: row 3 is valid
Validation result from process ID-19073: row 4 is valid
Validation result from process ID-19074: row 5 is invalid
Validation result from process ID-19075: row 6 is valid
Validation result from process ID-19076: row 7 is valid
Validation result from process ID-19077: row 8 is valid
Validation result from process ID-19078: row 9 is valid
Validation result from process ID-19079: 7 out of 9 columns are valid
Validation result from process ID-19080: 7 out of 9 sub-grids are valid
There are 21 valid sub-grids, and thus the solution is invalid
```

Figure 6: Threads: multiFail

```
[17727626@lab221-b01 partB]$ ./mssv ../testFiles/multiFail.txt 10
Validation result from thread ID-3998418688: row 1 is invalid
Validation result from thread ID-3990025984: row 2 is valid
Validation result from thread ID-3981633280: row 3 is valid
Validation result from thread ID-3973240576: row 4 is valid
Validation result from thread ID-3964847872: row 5 is invalid
Validation result from thread ID-3956455168: row 6 is valid
Validation result from thread ID-3990025984: row 7 is valid
Validation result from thread ID-3813844736: row 8 is valid
Validation result from thread ID-3948062464: row 9 is valid
Validation result from thread ID-3939669760: 7 out of 9 columns are valid
Validation result from thread ID-3931277056: 7 out of 9 sub-grids are valid
There are 21 valid sub-grids, and thus the solution is invalid
```

4 README

4.1 Purpose

The program validates an input file containing a sudoku solution. There are two versions. One utilising processes and the other utilising threads.

4.2 Running the Program

To compile the C files into an executable format.

```
make
```

To run the program there are two options.

Option 1:

```
make run
```

This will only let you run with the preset parameters and they will need to be altered in the Makefile to test other *input files* and *delays*

```
INPUT: ../testFiles/specTest.txt
DELAY: 10
```

Option 2:

```
./mssv <inputFile> <delay>
```

Between each time the program is run, the following command should be entered and executed. This is to delete the logfile produced from an invalid test file.

```
make clean
```

4.3 Files

partA

- Makefile
- mssv.c
- mssv.h

partB

- Makefile
- mssv.c
- mssv.h

testFiles

- test files

References

[1] *Interprocess communication using POSIX Shared Memory in Linux*. URL: https://www.softprayog.in/programming/interprocess-communication-using-posix-shared-memory-in-linux.

[2] *POSIX Semaphores*. URL: https://www.softprayog.in/programming/posix-semaphores.

[3] *POSIX Threads Programming in C*. URL: https://www.softprayog.in/programming/posix-threads-programming-in-c.

[4] *POSIX Threads Synchronization in C*. URL: https://www.softprayog.in/programming/posix-threads-synchronization-in-c.

5 Appendices

5.1 Processes: mssv.c

```
#include "mssv.h"

int main (int argc, char* argv[])
{
    // Validate command line parameters
    validateUse(argc, argv);

    // Rename command line parameters
    char* inputFile = argv[1];
    int maxDelay = atoi(argv[2]);

    // Variables
    int i, pid, processNum, numbers[] = {0,0,0,0,0,0,0,0,0};
    Region* region;
    sem_t semMutex, semParent, *semaphores;

    // File Descriptors
    int buff1FD, buff2FD, counterFD, semFD, regionFD, resFD;

    // Shared memory pointers
    int *buff2Ptr, *countPtr, *resourceCount, (*buff1Ptr)[NINE][NINE];

    // Create shared memory
    initMemory( &buff1FD, &buff2FD, &counterFD, &semFD, &regionFD, &resFD);

    // Map shared memory to pointers
    mapMemory(&buff1FD, &buff2FD, &counterFD, &semFD, &regionFD, &resFD,
                &buff1Ptr, &buff2Ptr, &countPtr, &semaphores, &region,
                    &resourceCount);

    // Initialise semaphores
    if ((sem_init(&semMutex, 1, 1)== 1) ||(sem_init(&semParent, 1, 1) == 1))
    {
        fprintf(stderr, "Could not initialise semaphores\n");
        exit(1);
    }

    semaphores[0] = semMutex;
    semaphores[1] = semParent;

    // Initialise parameters
    *countPtr = 0;
    pid = -1;
    processNum = 0;

    // Read input file
    readFile(inputFile, NINE, NINE, buff1Ptr);

    // Parent aquires lock of resourceCount
    sem_wait(&(semaphores[1])); // Lock child

    *resourceCount = 0;

    // Create child processes for
    while( processNum < NUMPROCESSES && pid != 0 )
    {
        signal(SIGCHLD, SIG_IGN); // Kill zombie processNum-1
        pid = fork();

        // Allow the parent to increment shared variable count
        if ( pid > 0)
        {
            *resourceCount = *resourceCount + 1;
```

```
64          }
65          processNum++;
66      }
67
68      if( pid == 0) // Child process
69      {
70          childManager(region, semaphores, buff1Ptr, buff2Ptr, countPtr,
71                                  resourceCount,  processNum, numbers, maxDelay );
72      }
73      else if ( pid > 0) // Parent process
74      {
75          parentManager(region, semaphores, countPtr, resourceCount);
76
77          // Clean up shared memory
78          cleanMemory(&buff1Ptr, &buff2Ptr, &countPtr, &semaphores,
79                          &region, &resourceCount, buff1FD, buff2FD, counterFD,
80                                semFD, regionFD, resFD);
81      }
82      else // Unsuccessful child process creation attempt
83      {
84          fprintf(stderr, "Unable to create child processes. Please run \"killall mssv\"\n");
85      }
86  }
87
88  /*****************************************************************************/
89
90  /**
91   * Read the contents of the input file passed as a command line argument
92   * @param inputFile File to be read
93   * @param rows      Number of rows in matrix
94   * @param cols      Number of columns in matrix
95   * @param buffer    Matrix to store contents of input file
96   */
97  void readFile(char* inputFile, int rows, int cols, int (*buffer)[rows][cols])
98  {
99      FILE* inStrm;
100     int i, j, count = 0;
101
102     inStrm = fopen(inputFile, "r"); // Open file for reading
103
104     if (inStrm == NULL) // Check file opened correctly
105     {
106         perror("Error opening file for reading\n");
107         exit(1);
108     }
109
110     // Store contents of file in 2D array
111     for( i = 0; i < rows; i++ )
112     {
113         for ( j = 0; j < cols; j++ )
114         {
115             count += fscanf( inStrm, "%d", &(*(buffer))[i][j] );
116         }
117     }
118
119     fclose(inStrm); // Close file
120
121     if(count != NINE*NINE)
122     {
123         fprint(stderr, "Invalid number of parameters read from file\n");
124         exit(1);
125     }
126 }
127
128 /**
129  * Write the invalid regions to log file
130  * @param region Sub region
131  * @param format String to be written
```

```
132  */
133 void writeFile(Region* region, char* format)
134 {
135     char* filename = "logfile";
136     FILE* outFile;
137     int val;
138 
139     outFile = fopen(filename, "a"); // Open file for appending
140     if (outFile == NULL) // Check file opened correctly
141     {
142         perror("Error opening file for writing\n");
143         exit(1);
144     }
145 
146     fprintf(outFile, "process ID-%d: %s",region->pid, format);
147 
148     fclose(outFile); // Close file
149 }
150 
151 /**
152  * Set each index to zero
153  * @param numbers Array to be reset
154  */
155 void resetArray(int numbers[])
156 {
157     for (int i = 0; i < NINE; i++)
158     {
159         numbers[i] = 0;
160     }
161 }
162 
163 /**
164  * Check if the contents of the array has any value other than one
165  * @param  numbers Array to be checked
166  * @return         Status of array being valid or not
167  */
168 int checkValid(int numbers[])
169 {
170     for (int j = 0; j < NINE; j++)
171     {
172         if ( numbers[j] != 1)
173         {
174             return FALSE;
175         }
176     }
177 
178     return TRUE;
179 }
180 
181 /**
182  * Handles the routine for the parent process. Outputs the result to the screen
183  * @param region        Array containing each region struct
184  * @param semaphores    Array of all semaphores
185  * @param countPtr      Pointer to shared memory counter
186  * @param resourceCount Status of number of child processes executing
187  */
188 void parentManager(Region *region, sem_t *semaphores, int* countPtr,
189                         int* resourceCount)
190 {
191     char *type, *message;
192     sem_post(&(semaphores[1])); // Unlock child
193     int done = FALSE;
194     int position;
195 
196     while( !done ) // Wait for all children to finish executing
197     {
198         //printf("Parent Waiting for Children\n");
199         sem_wait(&(semaphores[1])); // Lock child
```

```
200         if ( *resourceCount == 0)
201         {
202             done = TRUE;
203         }
204         sem_post(&(semaphores[1])); // Unlock child
205 
206     }
207 
208     for(int ii = 0; ii < NUMPROCESSES; ii++)
209     {
210         sem_wait(&(semaphores[0])); //Lock mutex
211         if (region[ii].type == ROW)
212         {
213             position = region[ii].positionX;
214             type = "invalid";
215 
216             if (region[ii].valid == TRUE)
217             {
218                 type = "valid";
219             }
220             printf("Validation result from process ID-%d: row %d is %s\n",
221                                     region[ii].pid, position, type);
222         }
223         else if (region[ii].type == COL)
224         {
225             type = "column";
226             position = region[ii].positionX;
227             printf("Validation result from process ID-%d: %d out of 9 columns are valid\n"
228                         ,region[ii].pid, region[ii].positionX);
229         }
230         else
231         {
232             type = "sub-grid";
233             position = region[ii].positionX;
234 
235             printf("Validation result from process ID-%d: %d out of 9 sub-grids are valid\n"
236                         ,region[ii].pid, region[ii].positionX);
237 
238         }
239 
240         sem_post(&(semaphores[0])); //Unlock mutex
241     }
242 
243 
244   if (*countPtr == 27)
245   {
246       message = "valid";
247   }
248   else
249   {
250       message = "invalid";
251   }
252 
253   printf("There are %d valid sub-grids, and thus the solution is %s\n", *countPtr, message);
254 }
255 
256 
257 /**
258  * Routine for child processes. Check the validity of sub region.
259  * @param region        Sub-region struct for each process
260  * @param semaphores    Array of semaphores
261  * @param buff1Ptr      Pointer to buffer1 in shared memory
262  * @param buff2Ptr      Pointer to buffer2 in shared memory
263  * @param countPtr      Pointer to counter in shared memory
264  * @param resourceCount Pointer to resourceCount in shared memory
265  * @param processNum    Child process number
266  * @param numbers       Array of numbers to check validity of sub region
267  * @param maxDelay      Delay for each process
```

```
268  */
269 void childManager(Region *region, sem_t *semaphores, int (*buff1Ptr)[NINE][NINE],
270                    int *buff2Ptr, int* countPtr, int* resourceCount,
271                        int processNum, int *numbers, int maxDelay )
272 {
273     char format[500];
274     int numValid, comma = 0;
275 
276     // Generate random maxDelay
277     srand((unsigned) getpid() );
278 
279     maxDelay = ( rand() % maxDelay ) + 1;
280 
281     if( processNum <= 9) // Check a row in buffer1
282     {
283         for (int i = 0; i < NINE; i++)
284         {
285             // Update numbers array
286             numbers[((*buff1Ptr)[processNum-1][i])-1]++;
287         }
288 
289         sleep(maxDelay); // Sleep
290         sem_wait(&(semaphores[0])); //Lock mutex
291 
292         // Update region struct
293         region[processNum-1].type = ROW;
294         region[processNum-1].positionX = processNum;
295         region[processNum-1].pid = getpid();
296         region[processNum-1].valid = checkValid(numbers);
297 
298         numValid = 0;
299         if (region[processNum-1].valid == TRUE)
300         {
301             numValid = 1;
302         }
303         else // Write to log file
304         {
305             sprintf(format, "row %d is invalid\n", processNum);
306             writeFile(&(region[processNum-1]), format);
307         }
308 
309         buff2Ptr[processNum-1] = numValid; // Update buffer2
310 
311         *countPtr = *countPtr + numValid; // Update counter
312 
313         sem_post(&(semaphores[0])); // Unlock child
314 
315     }
316     else if(processNum == 10) // Check all columns
317     {
318         sprintf(format, "column ");
319 
320         int validCol = 0;
321         for ( int nn = 0; nn < NINE; nn++) // Iterate through each column
322         {
323             for(int ii = 0; ii < NINE; ii++) // Iterate through each row
324             {
325                 numbers[(*buff1Ptr)[ii][nn]-1]++; // Update numbers array
326             }
327 
328             if ( checkValid( numbers) == TRUE )
329             {
330               validCol++;
331             }
332             else
333             {
334                 if (comma == 0)
335                 {
```

```
336                     comma = 1;
337                     sprintf(format + strlen(format), "%d", nn+1);
338                 }
339                 else
340                 {
341                     sprintf(format + strlen(format), ", %d ", nn+1);
342                 }
343             }
344 
345             resetArray(numbers);
346         }
347 
348         sleep(maxDelay);
349         if (validCol == 8)
350         {
351             sprintf(format + strlen(format), " is invalid\n");
352         }
353         else
354         {
355             sprintf(format + strlen(format), "are invalid\n");
356         }
357         sem_wait(&(semaphores[0])); //Lock mutex
358 
359         // Update region struct
360         region[processNum-1].type = COL;
361         region[processNum-1].positionX = validCol;
362         region[processNum-1].pid = getpid();
363         if(validCol != 9)
364         {
365             writeFile(&(region[processNum-1]), format);
366         }
367 
368         numValid = region[processNum-1].positionX;
369 
370         buff2Ptr[processNum-1] = validCol; // Update buffer2
371 
372         *countPtr = *countPtr + validCol; // Update counter
373 
374         sem_post(&(semaphores[0])); // Unlock mutex
375     }
376     else if( processNum == 11) // Check sub-grids
377     {
378         sprintf(format, "sub-grid ");
379 
380         int validSub = 0;
381 
382         // Iterate through each of the 9 3x3 sub-grid
383         for ( int jj = 0; jj < 3; jj++)
384         {
385             for (int kk = 0; kk < 3; kk++)
386             {
387                 for (int ll = jj*3; ll < jj*3+3; ll++)
388                 {
389                     for (int mm = kk*3; mm < kk*3+3; mm++)
390                     {
391                         // Update numbers array
392                     numbers[(*buff1Ptr)[ll][mm]-1]++;
393                   }
394               }
395 
396               if ( checkValid(numbers) == TRUE )
397               {
398                     validSub++;
399               }
400               else // Update string for log file
401               {
402                   if (comma == 0)
403                   {
```

```
404                         comma = 1;
405                         sprintf(format+strlen(format), "[%d..%d, %d..%d]",
406                                 jj*3+1, jj*3+3, kk*3+1, kk*3+3);
407                     }
408                     else
409                     {
410                         sprintf(format+strlen(format), ", [%d..%d, %d..%d] ",
411                                 jj*3+1, jj*3+3, kk*3+1, kk*3+3);
412                     }
413                 }
414                 resetArray(numbers);
415             }
416 
417         }
418 
419         sleep(maxDelay);
420         if (validSub == 8)
421         {
422             sprintf(format+strlen(format), " is invalid\n");
423         }
424         else
425         {
426             sprintf(format+strlen(format), "are invalid\n");
427         }
428         sem_wait(&(semaphores[0])); //Lock mutex
429 
430         // Update region struct
431         region[processNum-1].type = SUB_REGION;
432         region[processNum-1].positionX = validSub;
433         region[processNum-1].pid = getpid();
434 
435         if(validSub != 9) // Write to log file
436         {
437             writeFile(&(region[processNum-1]), format);
438         }
439 
440         buff2Ptr[processNum-1] = validSub; // Update buffer 2
441 
442         *countPtr = *countPtr + validSub; // Update counter
443 
444         sem_post(&(semaphores[0])); // Unlock mutex
445     }
446 
447     // Child signals it is finished by incremented resourceCount
448     sem_wait(&(semaphores[1])); // Lock child
449         *resourceCount = *resourceCount - 1;
450     sem_post(&(semaphores[1])); // Unlock child
451 
452 }
453 
454 /**
455  * Initalise shared memory constructs
456  * @param buff1FD    File descriptor for buffer1
457  * @param buff2FD    File descriptor for buffer2
458  * @param counterFD  File descriptor for counter
459  * @param semFD      File descriptor for semaphores
460  * @param regionFD   File descriptor for regions
461  * @param resFD      File descriptor for resourceCount
462  */
463 void initMemory( int* buff1FD, int* buff2FD, int* counterFD, int* semFD,
464                  int* regionFD, int* resFD)
465 {
466 
467     // Create shared memory
468     *buff1FD = shm_open("buffer1", O_CREAT | O_RDWR, 0666);
469     *buff2FD = shm_open("buffer2", O_CREAT | O_RDWR, 0666);
470     *counterFD = shm_open("counter", O_CREAT | O_RDWR, 0666);
471     *semFD = shm_open("semaphores", O_CREAT | O_RDWR, 0666);
```

```
472     *regionFD = shm_open("region", O_CREAT | O_RDWR, 0666);
473     *resFD = shm_open("resources", O_CREAT | O_RDWR, 0666);
474 
475     // Check shared memory was created correctly
476     if ( *buff1FD == -1 || *buff2FD == -1 || *counterFD == -1 || *semFD == -1 ||
477             *regionFD == -1 || *resFD == -1 )
478     {
479         fprintf( stderr, "Error creating shared memory blocks\n" );
480         exit(1);
481     }
482 
483     // Set size of shared memory constructs
484     if ( ftruncate(*buff1FD, sizeof(int) * NINE * NINE) == -1 )
485     {
486         fprintf( stderr, "Error setting size of buffer1" );
487         exit(1);
488     }
489 
490     if ( ftruncate(*buff2FD, sizeof(int) * NUMPROCESSES) == -1 )
491     {
492         fprintf( stderr, "Error setting size of buffer2" );
493         exit(1);
494     }
495 
496     if ( ftruncate(*counterFD, sizeof(int)) == -1 )
497     {
498         fprintf( stderr, "Error setting size of counter" );
499         exit(1);
500     }
501 
502     if ( ftruncate(*semFD, sizeof(sem_t) * 2 ) == -1 )
503     {
504         fprintf( stderr, "Error setting size of semaphores" );
505         exit(1);
506     }
507 
508     if ( ftruncate(*regionFD, sizeof(Region)*NUMPROCESSES) == -1 )
509     {
510         fprintf( stderr, "Error setting size of regions" );
511         exit(1);
512     }
513 
514     if ( ftruncate(*resFD, sizeof(int)) == -1 )
515     {
516         fprintf( stderr, "Error setting size of resourceCount" );
517         exit(1);
518     }
519 }
520 
521 /**
522  * Map shared memory to addresses
523  * @param buff1FD       File descriptor for buffer1
524  * @param buff2FD       File descriptor for buffer2
525  * @param counterFD     File descriptor for counter
526  * @param semFD         File descriptor for semaphores
527  * @param regionFD      File descriptor for regions
528  * @param resFD         File descriptor for resourceCount
529  * @param buff1Ptr      Pointer to buffer1 in shared memory
530  * @param buff2Ptr      Pointer to buffer2 in shared memory
531  * @param countPtr      Pointer to counter in shared memory
532  * @param semaphores    Array of semaphores
533  * @param region        Array of region structs
534  * @param resourceCount Pointer to resourceCount in shared memory
535  */
536 void mapMemory(int* buff1FD, int* buff2FD, int* counterFD, int* semFD,
537                 int* regionFD, int* resFD, int (**buff1Ptr)[NINE][NINE], int (**buff2Ptr),
538                     int** countPtr, sem_t** semaphores, Region** region, int**
    resourceCount)
```

```
539 {
540     // Memory mapping
541     *buff2Ptr = (int*) mmap(NULL, sizeof(int)*NINE*NINE,
542                 PROT_READ | PROT_WRITE, MAP_SHARED, *buff2FD, 0);
543     *buff1Ptr = mmap(NULL, sizeof(int)*NUMPROCESSES,
544                 PROT_READ | PROT_WRITE, MAP_SHARED, *buff1FD, 0);
545     *countPtr = (int*) mmap(NULL, sizeof(int),
546                 PROT_READ | PROT_WRITE, MAP_SHARED, *counterFD, 0);
547     *semaphores = mmap(NULL, sizeof(sem_t) * 2,
548                 PROT_READ | PROT_WRITE, MAP_SHARED, *semFD, 0);
549     *region = mmap(NULL, sizeof(Region)*NUMPROCESSES,
550                 PROT_READ | PROT_WRITE, MAP_SHARED, *regionFD, 0);
551     *resourceCount = mmap(NULL, sizeof(int),
552                 PROT_READ | PROT_WRITE, MAP_SHARED, *resFD, 0);
553 }
554
555 /**
556  * Validate command line parameters
557  * @param argc number of parameters
558  * @param argv command line parameters
559  */
560 void validateUse(int argc, char* argv[])
561 {
562     // Ensure correct number of command line parameters
563     if (argc != 3)
564     {
565         printf("Ensure there are the correct number of parameters\n");
566         exit(1);
567     }
568
569     // Ensure maxDelay is positive
570     if ( atoi(argv[2]) < 0)
571     {
572         printf("The maxDelay must be non-negative\n");
573         exit(1);
574     }
575 }
576
577 /**
578  * Close and destroy, semaphores and shared memory constructs
579  * @param buff1Ptr      Pointer to buffer1 in shared memory
580  * @param buff2Ptr      Pointer to buffer2 in shared memory
581  * @param countPtr      Pointer to counter in shared memory
582  * @param semaphores    Array of semaphores
583  * @param region        Array of region structs
584  * @param resourceCount Pointer to resourceCount in shared memory
585  * @param buff1FD       File descriptor for buffer1
586  * @param buff2FD       File descriptor for buffer2
587  * @param counterFD     File descriptor for counter
588  * @param semFD         File descriptor for semaphores
589  * @param regionFD      File descriptor for regions
590  * @param resFD         File descriptor for resourceCount
591  */
592 void cleanMemory(int (**buff1Ptr)[NINE][NINE], int **buff2Ptr, int** countPtr,
593                  sem_t **semaphores, Region **region,
594                     int** resourceCount, int buff1FD, int buff2FD,
595                         int counterFD, int semFD, int regionFD,
596                             int resFD )
597 {
598     // Close semaphores
599     sem_close(&((*semaphores)[0]));
600     sem_close(&((*semaphores)[1]));
601
602     // Destroy semaphores
603     sem_destroy(&((*semaphores)[0]));
604     sem_destroy(&((*semaphores)[1]));
605
606     // Clean up shared memory
```

```
607     shm_unlink("buffer1");
608     shm_unlink("buffer2");
609     shm_unlink("counter");
610     shm_unlink("semaphores");
611     shm_unlink("region");
612     shm_unlink("resources");
613 
614     // Close file descriptors
615     close(buff1FD);
616     close(buff2FD);
617     close(counterFD);
618     close(semFD);
619     close(regionFD);
620     close(resFD);
621 
622     // Unmap memory
623     munmap(*buff1Ptr, sizeof(int)*NINE*NINE);
624     munmap(*buff2Ptr, sizeof(int)*NUMPROCESSES);
625     munmap(*countPtr, sizeof(int));
626     munmap(*semaphores, sizeof(sem_t)*2);
627     munmap(*region, sizeof(Region)*NUMPROCESSES);
628     munmap(*resourceCount, sizeof(int));
629 
630     // Unlink shared memory constructs
631     shm_unlink("buffer1");
632     shm_unlink("buffer2");
633     shm_unlink("counter");
634     shm_unlink("semaphores");
635     shm_unlink("region");
636     shm_unlink("resources");
637 
638 }
```

5.2 Processes: mssv.h

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <fcntl.h>
#include <semaphore.h>
#include <sys/mman.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <time.h>

#define NINE 9
#define SUB 3
#define NUMPROCESSES 11
#define FALSE 0
#define TRUE !FALSE

typedef enum {ROW, COL, SUB_REGION} Region_Type;

typedef struct
{
    Region_Type type;
    int positionX;
    pid_t pid;
    int valid;
} Region;


void readFile(char* inputFile, int rows, int cols, int (*buffer)[rows][cols]);
void writeFile(Region* region, char* format);
void resetArray(int numbers[]);
int checkValid(int numbers[]);
void parentManager(Region *region, sem_t *semaphores, int* countPtr,
                        int* resourceCount );
void childManager(Region *region, sem_t *semaphores,
                    int (*buff1Ptr)[NINE][NINE], int *buff2Ptr, int* countPtr,
                        int* resourceCount, int processNum, int *numbers,
                            int maxDelay );
void initMemory( int* buff1FD, int* buff2FD, int* counterFD, int* semFD,
                    int* regionFD, int* resFD);
void mapMemory(int* buff1FD, int* buff2FD, int* counterFD, int* semFD,
                    int* regionFD, int* resFD, int (**buff1Ptr)[NINE][NINE],
                        int (**buff2Ptr), int** countPtr, sem_t** semaphores,
                            Region** region, int** resourceCount);
void validateUse(int argc, char* argv[]);
void cleanMemory(int (**buff1Ptr)[NINE][NINE], int **buff2Ptr, int** countPtr,
                    sem_t **semaphores, Region **region, int** resourceCount,
                         int buff1FD, int buff2FD, int counterFD, int semFD,
                            int regionFD, int resFD);
```

5.3 Processes: Makefile

```
# Makefile For Sudoku Solution Validator
# COMP2006 Assignment
# Last Modified: 12/04/17
# Jordan Yeo - 17727626

# MAKE VARIABLES
EXEC1 = mssv
OBJ1 = mssv.o
CFLAGS = -std=c99 -pthread -D _XOPEN_SOURCE=500 -lrt
CC = gcc
INPUT = ../testFiles/specTest.txt
DELAY = 1

# RULES + DEPENDENCIES
$(EXEC1) : $(OBJ1)
  $(CC) $(OBJ1) -o $(EXEC1) $(CFLAGS)

mssv.o : mssv.c mssv.h #fileIO.h
  $(CC) -c mssv.c $(CFLAGS)

clean:
  rm -f $(EXEC1) $(OBJ1) logfile

run:
  ./$(EXEC1) $(INPUT) $(DELAY)
```

5.4 Threads: mssv.c

```c
#include "mssv.h"

pthread_mutex_t mutex; // Mutex
pthread_cond_t use;  // condition for if the global variable is in use

int **buff1, *buff2, *counter, maxDelay, inUse;
Region *regions;

int main (int argc, char* argv[])
{
    // Validate command line parameters
    validateUse(argc, argv);

    // Rename command line parameters
    char* inputFile = argv[1];
    maxDelay = atoi(argv[2]);

    // Variables
    pthread_t threads[11];

    // Allocate  memory
    initMemory( &buff1, &buff2, &counter, &regions);

    *counter = 0;
    // Read input file
    readFile(inputFile, NINE, NINE, &buff1);

    // Initialise mutex and condition
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&use, NULL);
    inUse = 0;

    // Create threads
    for(int i = 0; i < NUMTHREADS; i++)
    {
        if ( i < NINE) // Initialise region struct for row threads
        {
            regions[i].type = ROW;
        }
        else if( i == NINE) // Initialise region struct for columns thread
        {
            regions[i].type = COL;
        }
        else // Initialise region struct for sub-grids thread
        {
            regions[i].type = SUB_GRID;
        }

        regions[i].position = i;
        resetArray(regions[i].numbers);
        // Create thread
        pthread_create(&(threads[i]), NULL, childManager, &(regions[i]));
        inUse++;
    }

    parentManager(); // Parent logic

    cleanMemory(); // Clean up malloc'd memory

}

/*****************************************************************************/

/**
 * Initalise memory constructs
 * @param buff1    buffer1 2D array
```

```
 * @param buff2    buffer2 1D array
 * @param counter  counter variable
 * @param regions  Region struct 1D array
 */
void initMemory(int*** buff1, int** buff2, int** counter, Region** regions)
{
    // Initialise
    *buff1 = (int**) malloc(sizeof(int*)* NINE);
    for (int i = 0; i < NINE; i++)
    {
        (*buff1)[i] = (int*) malloc(sizeof(int)* NINE);
    }
    *buff2 = (int*) malloc(sizeof(int)* NUMTHREADS);
    *counter = (int*) malloc(sizeof(int));
    *regions = (Region*) malloc(sizeof(Region)* NUMTHREADS);
}


/**
 * Free malloc'd memory and destroy mutex and conditions
 */
void cleanMemory()
{
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&use);
    for (int i = 0; i < NINE; i++)
    {
        free(buff1[i]);
    }
    free(buff1);
    free(buff2);
    free(counter);
    free(regions);
}


/**
 * Read the contents of the input file passed as a command line argument
 * @param inputFile File to be read
 * @param rows      Number of rows in matrix
 * @param cols      Number of columns in matrix
 * @param buffer    Matrix to store contents of input file
 */
void readFile(char* inputFile, int rows, int cols, int***buffer )
{
    FILE* inStrm;
    int i, j, count = 0;

    inStrm = fopen(inputFile, "r"); // Open file for reading

    if (inStrm == NULL) // Check file opened correctly
    {
        perror("Error opening file for reading\n");
        exit(1);
    }

    // Store contents of file in 2D array
    for( i = 0; i < rows; i++ )
    {
        for ( j = 0; j < cols; j++ )
        {
            count += fscanf( inStrm, "%d", &(*(buffer))[i][j] );
        }
    }
    fclose(inStrm); // Close file

    if(count != NINE*NINE)
    {
```

```
135         fprint(stderr, "Invalid number of parameters read from file\n");
136         cleanMemory();
137         exit(1);
138     }
139 }
140
141 /**
142  * Write the invalid regions to log file
143  * @param region Sub region
144  * @param format String to be written
145  */
146 void writeFile(Region* region, char* format)
147 {
148     char* filename = "logfile";
149     FILE* outFile;
150     int val;
151
152     outFile = fopen(filename, "a"); // Open file for appending
153     if (outFile == NULL) // Check file opened correctly
154     {
155         perror("Error opening file for writing\n");
156         exit(1);
157     }
158
159     fprintf(outFile, "thread ID-%d: %s",region->tid, format);
160
161     fclose(outFile); // Close file
162 }
163
164 /**
165  * Set each index to zero
166  * @param numbers Array to be reset
167  */
168 void resetArray(int numbers[])
169 {
170     for (int i = 0; i < NINE; i++)
171     {
172         numbers[i] = 0;
173     }
174 }
175
176 /**
177  * Check if the contents of the array has any value other than one
178  * @param  numbers Array to be checked
179  * @return         Status of array being valid or not
180  */
181 int checkValid(int numbers[])
182 {
183     for (int j = 0; j < NINE; j++)
184     {
185         if ( numbers[j] != 1)
186         {
187             return FALSE;
188         }
189     }
190
191     return TRUE;
192 }
193
194 /**
195  * Handles the routine for the parent. Outputs the result to the screen
196  * @param threads ID of child threads
197  */
198 void parentManager()
199 {
200     char *type, *message;
201     int done = FALSE;
202     int position;
```

```
203
204     pthread_mutex_lock(&mutex); // Lock mutex
205     while ( inUse > 0 ) // Wait while children are executing
206     {
207         pthread_cond_wait(&use, &mutex);
208     }
209
210     pthread_cond_signal(&use);
211     pthread_mutex_unlock(&mutex); // Unlock mutex
212
213     for(int ii = 0; ii < NUMTHREADS; ii++)
214     {
215         pthread_mutex_lock(&mutex); // Lock mutex
216
217         if (regions[ii].type == ROW)
218         {
219             type = "row";
220             position = regions[ii].position;
221             if ( regions[ii].valid == TRUE)
222             {
223                 printf("Validation result from thread ID-%u: %s %d is valid\n",
224                                         regions[ii].tid,type, position+1);
225             }
226             else
227             {
228                 printf("Validation result from thread ID-%u: %s %d is invalid\n",
229                           regions[ii].tid, type, position+1);
230             }
231         }
232         else if (regions[ii].type == COL)
233         {
234             type = "column";
235             printf("Validation result from thread ID-%u: %d out of 9 columns are valid\n"
236                     ,regions[ii].tid, regions[ii].count);
237         }
238         else
239         {
240             type = "sub-grid";
241             printf("Validation result from thread ID-%u: %d out of 9 sub-grids are valid\n"
242                     ,regions[ii].tid, regions[ii].count);
243         }
244
245         pthread_mutex_unlock(&mutex);
246     }
247
248
249   if (*counter == 27)
250   {
251       message = "valid";
252   }
253   else
254   {
255       message = "invalid";
256   }
257
258   printf("There are %d valid sub-grids, and thus the solution is %s\n", *counter, message);
259
260 }
261
262 /**
263  * Routine for child threads. Check the validity of sub region.
264  * @param args Void pointer to Region struct for the child
265  */
266 void* childManager(void* args )
267 {
268     char format[500];
269     int numValid;
270     Region* region = ((Region*)(args));
```

```
271     int threadNum = region->position;
272     int comma = 0;
273     int delay;
274 
275     // Generate random maxDelay
276     srand((unsigned) pthread_self() );
277     delay = ( rand() % delay ) + 1;
278 
279     if( region->type == ROW ) // Check row in buffer1
280     {
281 
282         // Check rows
283         for (int i = 0; i < NINE; i++)
284         {
285             // Update numbers array
286             region->numbers[((buff1)[threadNum][i])-1]++;
287         }
288 
289         sleep(delay); // Sleep
290         pthread_mutex_lock(&mutex); // Lock mutex
291 
292         // Update region struct
293         region->tid = pthread_self();
294         region->valid = checkValid(region->numbers);
295 
296         // Update buffer2
297         numValid = 0;
298         if (region->valid == TRUE)
299         {
300             numValid = 1;
301             region->count = numValid;
302         }
303         else // Write to log file
304         {
305             region->count = numValid;
306             sprintf(format, "row %d is invalid\n", threadNum+1);
307             writeFile((region), format);
308         }
309 
310         buff2[threadNum] = numValid; // Update buffer2
311 
312         *counter = *counter + numValid; // Update counter
313 
314         pthread_mutex_unlock(&mutex); // Unlock mutex
315 
316     }
317     else if( region->type == COL ) // Check all columns
318     {
319         sprintf(format, "column ");
320         int validCol = 0;
321         for ( int nn = 0; nn < NINE; nn++) // Iterate through each column
322         {
323             for(int ii = 0; ii < NINE; ii++) // Iterate through each row
324             {
325                 // Update numbers array
326                 region->numbers[((buff1)[ii][nn])-1]++;
327             }
328 
329             // Check if the column is valid
330             if ( checkValid( region->numbers) == TRUE )
331             {
332               validCol++;
333             }
334             else
335             {
336                 if (comma == 0)
337                 {
338                     comma = 1;
```

```
339                     sprintf(format + strlen(format), "%d", nn+1);
340                 }
341                 else
342                 {
343                     sprintf(format + strlen(format), ", %d ", nn+1);
344                 }
345             }
346             resetArray(region->numbers);
347         }
348 
349         sleep(delay);
350         if (validCol == 8)
351         {
352             sprintf(format + strlen(format), " is invalid\n");
353         }
354         else
355         {
356             sprintf(format + strlen(format), "are invalid\n");
357         }
358         pthread_mutex_lock(&mutex); // Lock mutex
359 
360         // Update region struct
361         region->count = validCol;
362         region->tid = pthread_self();
363         if(validCol != 9)
364         {
365             writeFile((region), format);
366         }
367 
368         numValid = region->count;
369 
370         buff2[threadNum] = validCol; // Update buffer2
371 
372         *counter = *counter + validCol; // Update counter
373 
374         pthread_mutex_unlock(&mutex); // Unlock mutex
375     }
376     else if( region->type == SUB_GRID ) // Check all sub-grids
377     {
378         sprintf(format, "sub-grid ");
379 
380         int validSub = 0;
381 
382         // Iterate through each of the 9 3x3 sub-grid
383         for ( int jj = 0; jj < 3; jj++)
384         {
385             for (int kk = 0; kk < 3; kk++)
386             {
387                 for (int ll = jj*3; ll < jj*3+3; ll++)
388                 {
389                     for (int mm = kk*3; mm < kk*3+3; mm++)
390                     {
391                         // Update numbers array
392                         region->numbers[((buff1)[ll][mm])-1]++;
393                     }
394                 }
395 
396                 if ( checkValid(region->numbers) == TRUE )
397                 {
398                     validSub++;
399                 }
400                 else // Update string for log file
401                 {
402                     if (comma == 0)
403                     {
404                         comma = 1;
405                         sprintf(format+strlen(format), "[%d..%d, %d..%d]",
406                                 jj*3+1, jj*3+3, kk*3+1, kk*3+3);
```

```
407                     }
408                     else
409                     {
410                         sprintf(format+strlen(format), ", [%d..%d, %d..%d]",
411                                 jj*3+1, jj*3+3, kk*3+1, kk*3+3);
412                     }
413                 }
414                 resetArray(region->numbers);
415             }
416 
417         }
418 
419         sleep(delay);
420         if( validSub == 8)
421         {
422             sprintf(format+strlen(format), " is invalid\n");
423         }
424         else
425         {
426             sprintf(format+strlen(format), " are invalid\n");
427         }
428         pthread_mutex_lock(&mutex); // Lock mutex
429 
430         // Update region struct
431         region->count= validSub;
432         region->tid = pthread_self();
433 
434         if(validSub != 9)
435         {
436             writeFile((region), format);
437         }
438 
439         buff2[threadNum] = validSub; // Update buffer2
440 
441         *counter = *counter + validSub; // Update counter
442 
443         pthread_mutex_unlock(&mutex); // Unlock mutex
444 
445     }
446 
447     // Child signals it is finished by incremented resourceCount
448     pthread_mutex_lock(&mutex); // Lock mutex
449 
450     while( inUse == 0)
451     {
452         pthread_cond_wait(&use, &mutex);
453     }
454     inUse--; // Decrease count of child processes running
455     if (inUse == 0)
456     {
457         pthread_cond_signal(&use);
458     }
459     pthread_mutex_unlock(&mutex); // Unlock mutex
460     pthread_detach(pthread_self()); // Release resources
461 
462 }
463 
464 /**
465  * Validate command line parameters
466  * @param argc number of parameters
467  * @param argv command line parameters
468  */
469 void validateUse(int argc, char* argv[])
470 {
471     // Ensure correct number of command line parameters
472     if (argc != 3)
473     {
474         printf("Ensure there are the correct number of parameters\n");
```

```
475             exit(1);
476         }
477 
478         // Ensure maxDelay is positive
479         if ( atoi(argv[2]) < 0)
480         {
481             printf("The maxDelay must be non-negative\n");
482             exit(1);
483         }
484 }
```

5.5 Threads: mssv.h

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <fcntl.h>
#include <semaphore.h>
#include <sys/mman.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <time.h>
#include <pthread.h>

#define NINE 9
#define SUB 3
#define NUMTHREADS 11
#define FALSE 0
#define TRUE !FALSE

typedef enum {ROW, COL, SUB_GRID} Region_Type;

typedef struct
{
    Region_Type type;
    int position;
    pthread_t tid;
    int count;
    int valid;
    int numbers[NINE];

} Region;


void readFile(char* inputFile, int rows, int cols, int***buffer);
void writeFile(Region* region, char* format);
void resetArray(int numbers[]);
int checkValid(int numbers[]);
void parentManager(void);
void* childManager(void* args );
void initMemory( int*** buff1, int** buff2, int** counter, Region** regions);
void mapMemory(int* buff1FD, int* buff2FD, int* counterFD, int* semFD,
                    int* regionFD, int* resFD, int (**buff1Ptr)[NINE][NINE],
                        int (**buff2Ptr), int** countPtr, sem_t** semaphores,
                            Region** region, int** resourceCount);
void validateUse(int argc, char* argv[]);
void cleanMemory(void);
```

5.6 Threads: Makefile

```
# Makefile For Sudoku Solution Validator
# COMP2006 Assignment
# Last Modified: 12/04/17
# Jordan Yeo - 17727626

# MAKE VARIABLES
EXEC1 = mssv
OBJ1 = mssv.o
CFLAGS = -std=c99 -g -pthread -D _XOPEN_SOURCE=500 -lrt
CC = gcc
INPUT = ../testFiles/specTest.txt
DELAY = 10


# RULES + DEPENDENCIES
$(EXEC1) : $(OBJ1)
  $(CC) $(OBJ1) -o $(EXEC1) $(CFLAGS)

mssv.o : mssv.c mssv.h #fileIO.h
  $(CC) -c mssv.c $(CFLAGS)

#fileIO.o : fileIO.c fileIO.h
# $(CC) -c fileIO.c $(CFLAGS)

clean:
  rm -f $(EXEC1) $(OBJ1) logfile

run:
  ./$(EXEC1) $(INPUT) $(DELAY)
```